

NAS2-12961
7H-61-CR
021073

Performance of the Intel i860XP

King Lee¹

Report RND-001 January 1994



National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

ARC 275 (Rev Mar 93)

Performance of the Intel i860XP

King Lee¹

Report RND-001 January 1994

NAS Systems Development Branch
NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

klee@nas.nasa.gov

1. The mailing address for Lee is Computer Science Dept., California State University, Bakersfield, CA 93309. This work was supported through Contract NAS 2-12961, while the author was an employee of Computer Sciences Corporation.

The Performance of the Intel i860XPTM

King Lee

Abstract

The Intel i860XPTM is a second generation i860TM microprocessor used in the Intel Paragon supercomputer. The most important architectural change was the new pipelined quadload instruction. We compared the performance of simple kernels using and not using this new instruction and found that this instruction substantially improved performance. We also compared the performance of subroutines using the NAS860 library with the performance of compiler generated code. We found that in many cases the code generated by the compiler performs as well as code using the NAS860 library. However, the performance of code generated by the compiler can, in some cases, be significantly improved by simple transformations.

1 Introduction

The Paragon supercomputer manufactured by Intel is based on the i860XPTM microprocessor. Previous models of this line of supercomputers were based on the i860XRTM, and we expect improvement in performance in each node. This section will review the new features of the chip and evaluate, from a theoretical point of view, what performance gains we might hope to get. The most important architectural feature for scientific applications is the quadload instruction which can be used only under some circumstances. In the next section we measure the performance of simple kernels that use this instruction. Since the performance of the short kernels is not a good indication of the performance that one can get on realistic applications, the performances on several simple subroutines taken from an ADI Solve were measured in section 3.

We expect improvements in performance due to the following features of the chip:

- increased clock speed of 50 MHz instead of 40MHz;
- increased instruction and data cache sizes of 16K each instead of 8K;
- a redesigned memory subsystem that, among other features, allows for memory burst mode that increases memory bandwidth by 67 percent.
- a new pipelined quadload instruction that can load two adjacent double precision numbers every three clocks([6]). The lower address of the two double precision numbers must be aligned on a 128 bit boundary and the destination must be aligned floating point registers.
- better performance for the multiply instructions. For some values of the operands, the multiply unit stalls for two clocks (page C-2 in [1]).

For codes that use cache one might expect improvements in performance due to the following factors: (1) a cache miss will be filled more rapidly because of the increased memory bandwidth and (2) the miss ratio of the i860XPTM will decrease due to the larger cache. Since time to fill a cache line consists of a latency as well as transmission time, we should not expect the penalty for a cache miss to decrease 67 percent. Assuming an eight clock latency to fill cache (the same as on the i860XRTM), and a transmission time of six instead of eight clocks, we would expect the time to fill a line of cache to decrease by about 33 percent (20 percent due to increased speed, and 12 percent due to fewer clocks). Furthermore, a 16K cache might give a 6 percent miss ratio (see page 486 of [2]). Therefore, for programs that work out

of cache, the increased bandwidth will improve the performance of 6 percent of the instructions. We point out that the actual fraction of instructions involving memory operations and the actual miss ratio vary greatly from program to program and some scientific programs may have a higher miss ratio.

The i860TM microprocessor has a pipelined load instruction that bypasses cache. In a previous paper ([3]) the author showed that the pipelined load instruction could move data from memory to registers twice as fast as using the normal load instruction which loaded data into cache. In subsequent papers ([4], [5]) the author developed a library, the NAS860 library, of subroutines that simulated vector operations. On some vector operations the performance was limited by the memory bandwidth. The new pipelined quadload instruction on the i860XPTM has the potential to increase performance of those vector instructions. But this instruction is restricted to operands contiguous and aligned on quadword boundaries. The instruction cannot be used to access vectors with non unit stride. Sometimes it is not possible to have vectors aligned. For example, if we have vectors x, y, z and the loops

```

do 100 i = 1, 128
100      y(i) = a * x(i)
      ...
do 200 i = 1, 128
200      z(i) = x(i+1) + z(i)

```

the vector x cannot be aligned in both loops. However, this restriction is more apparent than real. If a vector is not aligned, we might handle the

first element of the vector as a special case and the rest of the vector will be aligned.

Even if we were able to use the pipelined quadload instruction, we should not expect dramatic improvements in performance for all subroutines. Consider the subroutine or vector operation **vm1** from the NAS860 library ([5]):

```
do i = 1 , n
    ylm(i) = a * x(i)
100      continue
```

On the i860XP™ the multiply pipeline can produce one result every two clocks. If we used the pipelined quad load instruction, the multiply unit would not be able to keep up so it would still take two clocks to perform one iteration of the loop. In this case the effect of increased memory bandwidth will be restricted to minimizing loop overhead for this subroutine.

The reader should keep in mind that the performance of the Paragon supercomputer depends on the communication performance between nodes as well as the performance on each node. The time spent on communication may hide the improvement in performance due the i860XP™. For example if one gets a 30 percent performance increase in the chip, and a 10 percent performance increase in the faster inter node communication system, and if half the time is spent communicating, then one might see a 20 percent improvement in performance.

2 Measurements

A number of subroutines were selected from the NAS860 library and the performances of those subroutines were measured. The time measurements were taken 400 times with vectors of length 160. The performance reported is based on the average value of the performance. Examination of the times measured showed a variation as much as a 20 percent variation between the shortest and largest time for most of the times. The value of 160 for the vector length was chosen because it was sufficiently large so we could obtain the asymptotic average performance and sufficiently small so that the all vectors could fit into one set of the cache.

Table 1 gives the results of the measurements. The column labeled **Quad** gives the measured performance on the i860XPTM using the pipelined quad load instruction (if applicable). The column labeled **No-Quad** gives the performance of the subroutine without using the pipelined quadload instruction. This column gives performance for accessing vectors with non-unit stride.

The first subroutine was **lv** which moves data from main memory into cache. The **Quad** version uses the quad pipelined load instruction to move data to registers and a quad store instruction to cache. The measured performance was 24.5 MWDS (million double precision words per second), whereas the theoretical peak performance would be 33.3 MWDS (two words every three clocks). Two factors may have caused us to get only 74 percent of the peak performance. First, there may have been some loop overhead that was not hidden. Second, an examination of the individual times that went into computing the average showed that there were several times (outliers) that were much larger than most of the other times. The cause of the outliers

Table 1: Performance of some subroutines

Mflops

Subroutine	Quad	No-Quad
lv	24.5	13.6
uv	21.5	9.1
vvm	na	21.2
vm1	12.3	11.0
vm2	20.1	8.8
vr	na	30.0 [†]
vr1a	26.0	24.0 [†]
vnrm1	34.5	30.2
va2	19.7	8.8
vva4	18.7	8.5
vvm5	na	36.5
vvm1	13.3	9.4
vvm1a	9.2	7.5
vvs1	14.0	10.4
vvm4c	5.5	4.2
vvm5c	6.3	4.3
vvs4c	5.6	4.3
vts5a	11.0	8.8
vts4	16.0	13.3

[†] assume 10 FLOPs per reciprocal ([4])

may be due to messages passing through the node and taking up memory bandwidth or memory refresh. Using the pipelined load instruction we measured 13.6 MWDS which is only 54 percent of the peak performance of 25 MWDS. It is not clear why we saw such a small fraction of the maximum theoretical bandwidth.

The subroutine **uv** moves data from cache to main memory. Using quadstore we get only 65 percent of the theoretical peak. Without using the quadload instruction, we get only 9.1 MWDS. Writing to memory is more complex than reading from memory. The words to be written to memory are placed in a write queue on the chip and then sent to memory when the bus is free. Storing into the on chip write queue may take an extra cycle under some circumstances.

The next routine **vvm** forms the outer product of two vectors in cache and stores the result in cache. It is equivalent to the following code:

```

do 100 i = 1, n
100      zlm(i) = xlm(i) * ylm(i).

```

The quad load and store instructions (which are also available on the i860XRTM) can be used to access the operands so, there is no need to use the pipelined quad load instruction available only on the i860XPTM. We achieve about 80 percent of the theoretical maximum performance on the i860XPTM. This suggests that the larger shortfall from peak performance in **lv** and **uv** may be due to an artifact of the memory subsystem.

The next subroutine, **vnrm1**, forms the norm of a vector in memory:

```

do 100 i = 1, n

```

```

      s = s + x(i) * x(i)
100 continue

```

We get respectable performance (34.5 MFLOPS) because there are two floating point operations for every memory operation. The subroutine **vr** forms the reciprocal of a vector from cache and **vr1a** forms the reciprocal of a vector from memory and stores it in cache. On the i860XPTM the reciprocal is formed by an initial reciprocal approximation, followed by Newton's iteration. Newton's iteration requires a total of nine floating point multiplies and additions. The performance of these subroutines was calculated on the basis of 10 floating point operation per result. Again these subroutines get respectable performance because of the large number of floating point operations compared to memory accesses. The definitions of the remaining subroutines are given in ([5]). Most of these subroutines involve one floating point operation and have one or more operands in main memory. For most of these subroutines, the performance is substantially less than what the peak performance would predict. However in almost all cases we get substantially better performance when using the quad load than when we do not.

In summary we still get best results when all operands are in cache. If one or more of the operands are in main memory, then we can still get a substantial fraction of peak performance by using the quad load or quad store. If that is not possible because we have a non-unit stride, we must use the double precision pipelined loads and stores which give less performance.

3 Subroutines from ADI Solve

The performance figures for short loops, such as those of the previous section, have two benefits: (1) they can point to weaknesses of the microprocessors and (2) they can help to give an upper bound on the performance that one can expect from realistic applications. In order to try to get a better estimate of the performance we can actually expect from realistic problems we measured the performance of several of the subroutines found in the ADI Solve supplied by Dr. Sisera Weeratunga ([4]). The performance of some of these subroutines was measured using the NAS860 library described in the previous section. For purposes of comparison the performance of subroutines was also measured using the PGI compiler with flags “-O4 -Knoieee -Mquad -Mvect”.

All the subroutines that we considered, except **systrd**, involved simple matrix operations with nested loops. The following table gives the measured performance for the subroutines using 120 by 120 matrices. The second column gives the performance using the NAS860 routines with pipelined quadload instructions, the third column gives the performance of the compiled subroutine, and the last column gives the performance of the compiled subroutine with a small amount of hand tuning.

The first subroutine, **l2norm**, computes the norm of a matrix:

```
do j = 1, n
  do i = 1, n
    nrm = nrm + a(i,j) * a(i,j)
  enddo
enddo
```

Subroutine	NAS860	Compiled	Compiled (transformed)
l2norm	38.0	18.3	na
comptp1	23.6	16.2	na
fajl	13.6	2.5	9.0
fail	7.4	4.3	6.6
fail (rev)	10.1	4.7	8.4
fbil	10.1	10.1	12.9
fbilr(rev)	12.2	10.1	6.5
systrd*	19.5	14.4	na

* assume 10 FLOPs per reciprocal ([4])

Table 2: Performance of ADI Subroutines (Mflops)

This subroutine essentially uses **vnrm1** repeatedly, and we can get high performance because each memory reference results in two floating point operations. An examination of the assembly listing of the compiled code suggests that the compiled code moves data into cache before computing the sum of the squares. The NAS860 does not bother moving data into cache; it moves data to registers and computes the computation there.

The second subroutine **comptp1** has code that is a little more complicated:

```

do j = 1, ncol
  do i = 1, nrow
    temp1 = c(i,j) - a(i,j)
    temp2 = c(i,j) - b(i,j)
    sum = sum + (temp1 * temp1)
    asum = asum + (temp2 * temp2)
  end do
end do

```

This subroutine is similar to the previous loop but it requires more memory accesses per floating point operation.

The next subroutine, **faj1**, manipulates some matrices.

```

do i = 1, nrow
  b(i,1) = ap2(i,1)
  c(i,1) = an(i,1)
  do j = 2, ncol-1
    a(i,j) = alm1 * as(i,j)
    b(i,j) = 1.0d+00 + alm1 * ap2(i,j)
    c(i,j) = alm1 * an(i,j)
  end do
  a(i,ncol) = as(i,ncol)
  b(i,ncol) = ap2(i,ncol)
enddo

```

Subroutine **faj1** (Original)

The inner loop in the original subroutine accesses the matrices by columns, and this means that we access the memory with non-unit stride. In preparing to use the NAS860 subroutines, we interchanged the order of loops and broke up the inner loops.

```

do i = 1, nrow
    b(i,1) = ap2(i,1)
enddo
do i = 1, nrow
    c(i,1) = an(i,1)
enddo
do j = 2, ncol-1
    do 100 i = 1, nrow
        a(i,j) = alm1 * as(i,j)
100        continue
    do 200 i = 1, nrow
        b(i,j) = 1.0d+00 + alm1 * ap2(i,j)
200        200
    do 300 i = 1, nrow
        c(i,j) = alm1 * an(i,j)
300        continue
    enddo
do i = 1, nrow
    a(i,ncol) = as(i,ncol)
enddo

```

```

do i = 1, nrow
    b(i,ncol) = ap2(i,ncol)
enddo

```

Subroutine **faj1** (Transformed)

Then each of the inner loops was replaced by a call to a NAS860 subroutine. When the original subroutine **faj1** was compiled we obtained a performance of 2.5 MFLOPS, whereas the NAS860 loops gave a performance of 13.6 MFLOPS. We then compiled the transformed loops and were surprised to see a performance of 9.0 MFLOPS. Evidently a loop interchange and decomposition of the loops can lead to substantial improvements in performance.

The next subroutine we considered was **fail**:

```

do j = 1, ncol
    b(j,1) = ap2(1,j)
    c(j,1) = ae(1,j)
enddo
do j = 1, ncol
    do i = 2,nrow-1
        a(j,i) = alm1*aw(i,j)
        b(j,i) = 1.0d+00 + alm1*ap2(i,j)
        c(j,i) = alm1*ae(i,j)
    end do
end do

```

```

do j = 1, ncol
  a(j,nrow) = aw(nrow,j)
  b(j,nrow) = ap2(nrow,j)
end do
  Subroutine fai1 (Original)

```

In the inner loop, the left hand side of the equation is indexed by column (and large stride) whereas the right side is indexed by row. It is not possible to have both accesses with stride one. To use the NAS860 library, the program was transformed to:

```

do j = 1, ncol
  b(j,1) = ap2(1,j)
enddo
do j = 1, ncol
  c(j,1) = ae(1,j)
enddo
do j = 1, ncol
  do i = 2,nrow-1
    a(j,i) = alm1 * aw(i,j)
  end do
  do i = 2,nrow-1
    b(j,i) = 1.0d+00 + alm1 * ap2(i,j)
  end do
  do i = 2,nrow-1
    c(j,i) = alm1 * ae(i,j)
  end do
end do

```

```

do j = 1, ncol
    a(j,nrow) = aw(nrow,j)
end do
do j = 1, ncol
    b(j,nrow) = ap2(nrow,j)
end do

```

Subroutine **fail** (Transformed)

As before, each inner loop was translated to a call to a NAS860 subroutine. The performances are given in Table 2. We see that the NAS860 subroutine gave the best performance, but the transformed compiled program gave about 85 percent, and the compiled original subroutine gave 60 percent of the performance of the NAS860 subroutine.

The original subroutine uses the quadload to load the vectors (inner loops have unit stride on the right), and non-unit stride on the store. Each of the inner loops required (1) a subroutine that would load a vector element, multiply that vector element by a constant, save that element in cache and (2) a subroutine to load an element from cache and store it in memory in two of the loops. Since there was more activity on loads, it was conjectured that if the loads were accessed with non-unit stride any stalls due to the multiply would be hidden by the longer memory access. To test this conjecture we interchanged the loops:

```

do j = 1, ncol
    b(j,1) = ap2(1,j)
    c(j,1) = ae(1,j)
enddo

```

```

do i = 2, nrow - 1
  do j = 1, ncol
    a(j,i) = alm1 * aw(i,j)
    b(j,i) = 1.0d+00 + alm1 * ap2(i,j)
    c(j,i) = alm1 * ae(i,j)
  end do
end do
do j = 1, ncol
  a(j,nrow) = aw(nrow,j)
  b(j,nrow) = ap2(nrow,j)
end do

```

Subroutine **reversed fai1**

The performance was measured for this subroutine, the transformed subroutine, and the NAS860 version. We see that there is a substantial improvement in the NAS860 and reverse transformed version. While we hoped for some improvement, the amount of improvement is surprising.

Next, we considered **fbi1**:

```

do i = 1, inl
  do j = 1, jnl
    d(j,i) = alm2 * rsd(i,j) * +(z/(1.0 + z ))*delf(i,j)
  end do
end do

```

Subroutine **fbi1**

The corresponding transformed subroutine is:

```

dpconst1 = z/ (1.0d+00 + z)
do i = 1, nrow
    do j = 1, ncol
        xlm(i) = alm2*rsd(i,j)
    end do
    do j = 1, ncol
        ylm(i) = dpconst1 *delf(i,j)
    end do
    do j = 1, ncol
        d(j,i) = xlm(i) + ylm(i)
    end do
end do
Subroutine transformed fbil

```

We measured the performance as above, and found that the compiled program matched the performance of the NAS860 subroutine, but the transformed subroutine surpassed the performance of the NAS860 subroutine. When we reversed the order of the loops, we found that the NAS860 subroutine increased in performance, whereas performance of the compiled code either decreased or remained the same. The compiler generates code that calls subroutines that are similar to the NAS860 subroutines. The behavior of compiled code cannot be explained since the source of the subroutines that the compiler called is not available.

The last subroutine is **systrd**, a tridiagonal solve. This subroutine is more complicated than the other subroutines and will not be discussed here (see [4] for details). The performance figures reported may be misleading. The subroutine **systrd** involved a reciprocal which could be computed rapidly

(see the performance of **vr** in Table 1), and we assigned 10 FLOPs for a division. It is usual to arbitrarily assign 3 FLOPs for division; doing so would have decreased the reported performance by 30 percent.

4 Conclusion

The i860XPTM has several new features that promise great improvement over the i860XRTM chip. The most important, from the point of view of scientific programming, is the new quadload instruction. In the best case this instruction would lead to a 67 percent increase in peak bandwidth. However we could only measure about 75 percent of the peak using the pipelined quadload instruction. The performance using the pipelined quadload instruction was about twice that of the pipelined load instruction. When operating out of cache, we were able to get over 80 percent of peak performance.

It is the impression of the author that the present PGI compiler showed a great improvement over the previous version of the compiler. Most of the compilers came within a factor of two of the NAS860 subroutines without modification. On many subroutines simple modifications improved performance markedly. We look forward to continued improvements in the compiler.

ACKNOWLEDGEMENT: The author thanks William Nitzberg for his careful reading and editorial suggestions of this paper.

References

- [1] Intel, *i860 64-Bit Microprocessor Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA., 1990.

- [2] Hennesy and Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 1990.
- [3] King Lee. *On the Floating Point Performance of the i860TM Microprocessor*, RNR-90-019, October, 1990. NAS, Ames Research Center, Moffett Field, CA 94035.
- [4] King Lee. *Achieving High Performance on the i860TM Microprocessor* RNR-91-029, October, 1991. NAS, Ames Research Center, Moffett Field, CA 94035
- [5] King Lee. *The NAS860 Library User's Manual* RND-93-003, March, 1993. NAS, Ames Research Center, Moffett Field, CA 94035
- [6] Shane Story, Intel Corp. Private communication.



RND TECHNICAL REPORT

Title: *Performance of the
Intel i860 XP*

Author(s): *King Lee*

Reviewers:

"I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document."

*Two reviewers
must sign.*

Signed:

Russell Carter

Name:

Russell Carter

Signed:

Bill Nitzberg

Name:

Bill Nitzberg

*After approval,
assign RND TR
number.*

Branch Chief:

Approved:

Bruce Bybee

Date:

1-24-94

TR Number:

RND-94-001

Important: Put this form as the last page in the published Tech Report.